Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 2086–2095

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

ALPS: A Methodology for Application-Level Communication Characterization of Parsec 2.1

Dominic Hillenbrand, Jie Tao, Matthias Balzer

Karlsruhe Institute of Technology

Abstract

This paper presents a methodology for analyzing communication of multi-threaded applications. Previous work relies on more or less accurate architectural models. Our measurement methodology has been designed to be completely architecture independent, since we want architects to have an undistorted view of the communication behavior. One part of our methodology is the concept of communication attribution which allows communication from underlying libraries, for example, to be attributed to application functionality. In this paper, we have applied our methodology to the Parsec benchmark suite. Our characterization shows how communication in Parsec scales, as we increase the number of threads from 2 to 256. Based on these results we have modeled communication growth as a power function and determined the characteristic exponents for each benchmark. Last but not least, our methodology enables researchers to track communication down to the source code.

Keywords: Cache-Communication, Multi-Core, Parsec, Benchmark, Performance Analysis, Simulation

1. Introduction

The performance of a micro-architectural design can be assessed by benchmarking. If the architecture targets only a limited application domain then it is sufficient to measure the performance with a small number or even a single application. This is especially common in embedded systems where a device often fullfills a single purpose. Nowadays this can also be said for dedicated server machines that run a single application, like for example a data base. Ironically, in non-trivial embedded systems such as today's smart phones a plethora of demanding applications are executed. To cope with computational demands, rising power consumption and Moore's law, the number of cores has been increasing in embedded systems, servers and even desktop machines. To assess the efficiency of general purpose chip multi-processors (CMPs) it is necessary to have more than a few applications for benchmarking. Hence a benchmark suite is necessary. Benchmark suites often originate from certain communities (e.g. embedded vs. high-performance computing) and make assumptions about the target architecture and input sets. For expressing parallelism they use different programming languages and paradigms (e.g. "pthreads", "OpenMP" or Intel Threading Blocks in Parsec). It is true that with time many architectural parameters change and benchmarks show signs of aging. However, there has been a relatively constant interest in what researchers want to measure. Architectural efficiency

Email address: {dominic.hillenbrand, jie.tao, matthias.balzer}@kit.edu (Dominic Hillenbrand, Jie Tao, Matthias Balzer)

Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.228

is well characterized by run-time and consumed energy. The latter is especially important in embedded systems and increasingly in data centers - where, for example, embedded processors are seriously considered in order to cut power costs. As the number of cores increase (with Moore's law) it is conceivable that in some applications the amount of communication will increase as well. Since communication has high costs in terms of time and power it is crucial to have a good understanding of its nature in an application. Therefore, we have analyzed the inter-thread communication behavior of the Parsec 2.1 benchmark suite.

2. Motivation

Earlier papers fix many architectural parameters and look at communication on a thread/core- or cache-level. It is obvious that the multitude of architectural design parameters chosen by other authors influence the results in many ways – which is perfectly desirable if a new design is evaluated.

Our goal – however - is to characterize the inherent communication behavior of Parsec with the least amount of architectural constrains. Our architecture independent methodology allows us to track communication inside threads - where we can locate communication down to the source code level. Additionally, we can attribute communication from non-application code (e.g. underlying libraries) to the application. These capabilities may be useful for: 1) compiler architects who want to optimize code for communication, 2) the design of application specific network on chip (NoCs) where, for example, ring- and mesh-topologies can be chosen for low- and high-communication demands, 3) selecting benchmark specific interconnect types and wire properties (low power wires vs. high power fat wires [1]) and 4) for different types of cache-coherency protocols. The architect may, for example, match the invalidation behavior of a given application with an optimized directory- or snoop protocol. Combinations of both may also be applicable.

This paper makes two contributions:

- An architecture independent, qualitative and quantitative characterization of communication in Parsec.
- A new methodology to assess communication in multi-threaded applications based on four transition types and communication attribution.

In the following section we provide background information about Parsec and its competitors.

3. Background and Related Work

"Historic" benchmarks for embedded systems such as MiBench [2] or MediaBench [3] looked at relatively small (in code size) single threaded kernels. Splash-2 [4], another aging (yet popular) benchmark suite from 1995, has its origin in the high performance computing community and makes outdated architectural assumptions (board-to-board communication = high latencies) which make it increasingly unsuitable for assessing the performance of modern CMP architectures. In [5] Parsec and Splash-2 have been compared. Parsec [6] in comparison is a relatively recent benchmark from Princeton University and Intel. It has been designed to exploit the characteristics of modern and upcoming CMPs. Its workloads have been chosen in anticipation of future scenarios. It remains to be seen if they will truly reflect those. The diverse Parsec workloads are grouped into the categories "applications" (total of 9) and "kernels" (total of 3). They are described in [6] and [7, 8]. A variety of parallel benchmark suites [9, 10, 11, 4] have been published before Parsec. They vary in age (architectural assumptions, programming models) and focus (community). In [6] the creators of Parsec argue that existing benchmarks (most prominently: SPLASH-2, SPEC CPU 2006 and OMP2001) are less suitable for exploring CMP designs. This paper covers Parsec but the methodology and tools can be applied to many other benchmarks as well.

The Parsec creators characterized the workloads in terms of parallelization overhead, working sets, locality, off-chip traffic, communication, computation ratio and last but not least sharing. They conducted their characterization using a simulator (PIN [12], 8 cores). A question raised by the authors was: What last-level cache size should be used? In [6] they settled with 4 MB and conceded that communication cannot be measured correctly due to cache capacity conflicts [6]. A later paper [8] uses greater cache sizes between 1–128 MB (4-way associative, 64 byte lines). Given the fact that some benchmarks scale their working set with the input set size, it would be necessary to scale the cache sizes accordingly. The authors have decided against scaling cache sizes in order to preserve comparability between their different measurements. Through their experiments the Parsec authors noticed high traffic volumes between

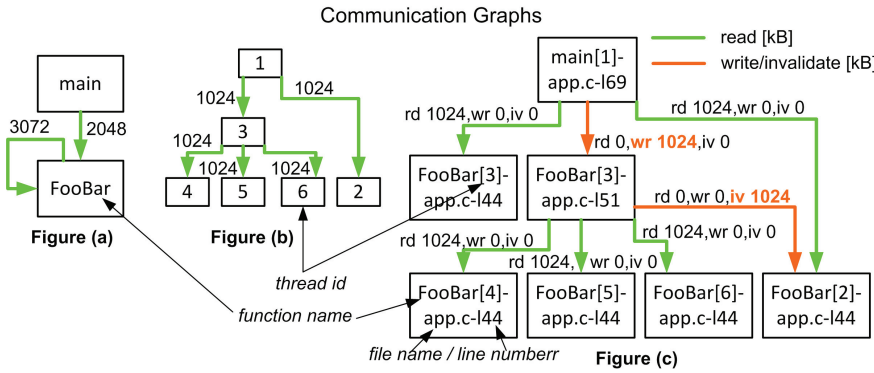

Figure 1: The recorded transitions of a multi-threaded test application (6 threads) have been used to generate these three different types of communication graphs. In (a) the nodes represent functions and in (b) thread ids. The edge weights in both graphs indicate the amount of "read" communication (later on referred to as CASE III in the algorithm – "other thread reads"). The last figure (c) is more detailed. The nodes are further broken down to source-code granularity and the edges additionally show the amount of data written and invalidated (CASE IV- "other thread writes").

threads. We provide quantitative data and use a different measuring methodology that avoids the aforementioned dilemma (cache size).

The authors of [13] characterized communication in Splash-2 and Parsec at thread-level granularity. Their simulated architecture has 32 cores, a 2-level cache hierarchy (4-way and 32-way associative, 64 byte line), a constant CPI (1.0) x86-processor and an infinite bandwidth zero-cycle crossbar. Due to simulation speed the "medium" input set of Parsec was chosen. The measurement methodology in [13] requires each thread to be mapped statically to one core. This was realized for 5 out of 13 Parsec workloads ("blackscholes", "canneal", "fluidanimate", "streamcluster", and "swaptions"). Furthermore, sharer tracking is imprecise (50% probability boundary). We include the initialization phase, use the "simlarge" input set (like [6]) and run Parsec (also in simulation) for 2–256 cores without fixing cache parameters. Both papers [6] and [13] characterize communication behavior at thread level granularity. Our methodology allows us to be more precise (down to instruction level). The following section describes the methodology used in prior papers and our own.

4. Methodology

There are multiple ways to assess communication in Parsec: First, Parsec can be run natively on existing machines. Second, a full-system simulator such as M5 [14] can be configured and extended to model a desired hardware architecture. Third, a comparatively new approach is to utilize FPGAs to accelerate simulation. There are mixed mode approaches [15] and purely FPGA-based accelerators which have been developed as part of the RAMP effort [16, 17, 18].

The approach chosen in this paper abstracts from the architecture. Instead of measuring the performance of actual architectures (e.g. in terms of miss rate for a specific cache), we track communication on instruction level. Whenever a memory location is read- or written to in a way that causes communication then we record one or more transitions. The concept of tracking different transitions (in total four) is the foundation for our measurements.

A transition encompasses the thread id (1), the instruction address (2), the function[1] *name (3) and the source code location (4).*

The transitions which we record are detailed enough to create communication graphs with varying levels of information (resulting in varying numbers of edges and nodes). After simulation, the recorded transitions can be used to build communication graphs (analogous to a call-graph of a single threaded application) – where the edge weights reflect the amount of data read, written or invalidated. Figure 1 shows three different communication graphs of the

[1] *"Functions" in this paper refer to compiled procedures (list of assembler instructions) which in turn correspond to procedures in the original source code (e.g. C/C++)*

same application – where a function "main" in thread 1 writes 1 MB of data. This data is read by a function "FooBar" in thread 2 and 3. Afterwards this data is overwritten by thread 3 which leads to invalidations in thread 2 – a previous sharer. Subsequent reads from threads 4—6 communicate with thread 3 – the last writer. The first two Figures 1 a) and b) show communication graphs where nodes represent a single parameter: function name or thread id. The edge weights in both graphs stand for the amount of data read.

Function names allow us to break-down communication in terms of application functionality – instead of threads.

The third graph c) is more detailed. Here, graph nodes stand for a specific line of code inside a function, inside a source file, executed by a thread. The edge weights in c) show the amount of data read, written and invalidated. The three communication graph types a)—c) are just examples. More graph types can be generated by combing one or more different parameters (here: thread id, file name, line number, and function name). The communication graph types allow different views of the same data set. For visualization, for example, it may be helpful to choose graph type a) when an application spawns hundreds of threads per function over its run-time. Thus, by pruning the thread ids from the graph it may become a lot easier to understand application-level communication on a function-level. Of course every communication graph can be pruned further by setting a threshold for edge weights (amount of data "read", "written" and "invalidated").With "line numbers" in graph nodes, we can track communication to a specific line of code which is invaluable in a detailed communication analysis.

4.1. Communication Attribution

An important contribution of this paper is the attribution of communication to an application – even when communication has been caused by an underlying library, for example.

In Figure 1, for example, prior memory contents are overwritten by a function called "FooBar" which resides in the file "app.c" (line 51). The actual writing, however, does take place in an otherwise unrelated library function included by "app.c". Through communication attribution we can thus map communication on lower levels (e.g. from libraries or operating system) to application-level. This crucial feature enables holistic application-level optimization and analysis of communication. In the following section, we will explain the algorithm.

4.2. Algorithm

We slice the memory space into configurable sized chunks. The size can, for example, be set to 1, 2, 4, 8, 16, 32 or even 512 bytes. The more fine grained the chunks are, the more memory is required to run the simulation. We initially provide data for 8- and for 32-byte granularity. 8 bytes match the machine word size (64bit) and is architecture neutral. 32 bytes are a common cache line size. Thus prefetching and false sharing effects are observable.

The pseudo-code of our algorithm is shown in Figure 2. The algorithm is invoked at run-time on every memory read- and write-request. The inputs "`read`", "`addr`" and "`size`" describe the parameters of the memory access. They indicate whether the access is a read, the memory address and the amount of data to be read or written. The instruction address "`iaddr`" (of the memory read/write instruction) is the granularity at which we track communication. This allows us to determine the enclosing function, the location in the source code (file name and line number) and whether the instruction belongs to the application. The unique thread id ("`tid`") tells us which thread is executing the memory instruction.

In the lines 1—3 we check if the memory instruction belongs to the application "`app_code`". If it does then we memorize "`iaddr`" under the current thread id. This mechanism allows us to attribute communication to the last caller on application level. The parameter "`app_code`" is pre-computed at instrumentation time using "`iaddr`".

In the lines 5—10 the algorithm retrieves information associated with the requested memory chunk - if available. Otherwise a new entry "`info`" is created. Each entry contains the instruction address "`iaddr`" of the last writer, its thread id and a list of sharers (initially empty).

The last writer and current reader/ writer are tuples with the designations "current" and "previous" (see line 13 and 14). Transitions take place between "`current`", "`previous`" and "`sharer`" for reads, writes and invalidations. The tuple definitions can, in principle be simplified or further refined. The more specific the tuple definitions become, the more transitions are likely recorded for an application run.

The current algorithm distinguishes four cases, I-IV (lines 16—39): first, a read by the same thread which is ignored (line 17 – no communication); second, a write by the same thread which invalidates sharers (line 20); third, a "read" from a different thread (line 26) and fourth a "write" from a different thread (line 31).

The Algorithm

```
1  if is_application_function(fptr) then
2     thread[tid].app_ptr = iaddr
3  endif
4
5  info = get_info_for(addr)
6  if not info then
7     if read return
8     info[addr] = thread[tid].app_ptr,tid, []
9     return
10 endif
11
12 current  = (thread[tid].app_ptr, tid)
13 previous = (info.ptr, info.tid)
14
15 match begin
16   // CASE I : Same thread read
17   case info.tid == tid and read return
18
19   // CASE II: Same thread write
20   case info.tid == tid and not read
21     foreach sharer in info.sharers
22       update_edge current [inv:+csize]→ sharer
23     end foreach
24     clear info.sharer
25
26   // CASE III: Other thread read
27   case info.tid != tid and read and not current in info.sharer
28     update_edge previous [read:+csize]→ current
29     info.sharer.add current
30
31   // CASE IV: Other thread write
32   case info.tid != tid and !read
33     foreach sharer in info.sharers
34       update_edge current [inv:+csize]→ sharer
35     end foreach
36     clear info.sharer
37     update_edge previous [write:+csize]→ current
38     info = current
39 end match
```

Figure 2: A memory access can match one of four cases. Some cases (II—IV) involve the creation of transitions. A case depends on prior information about the memory chunk (thread id) and the current memory request (read/write, thread id). Communication attribution is achieved by tracking the last-writer (instruction) on application-level.

The function "`update_edge`" creates a new transition. If the transition already exists then it only needs to add "csize" (chunk size) to the edge. A transition edge can be annotated with the amount of data read, written and invalidated (labels: "`read`", "`write`", "`inv`"). If a memory request has a size ("`size`") bigger then chunk size ("`csize`") then it is split into multiple requests of chunk size. This is not shown in Figure 2 in order to keep the focus on the core concepts.

In the following section we explain how commonly used communication patterns relate to our transition model.

4.3. Read-Only- / RW-Sharing, Migratory and Producer-Consumer Patterns

In [13] (Figure 1) the authors nicely describe four communication patterns: "read-only", "read/write", "producer/consumer" and "migratory". Some of the patterns have a close resemblance: the "read-only" pattern can be seen as a special case of the "producer/consumer" pattern where data is only communicated once. The "read/write"-pattern is a "producer/consumer" relationship among multiple threads.

The authors found that some of the aforementioned patterns are rarely seen in isolation within Parsec, especially "producer/consumer" vs. the more common multiple "producer/consumer".

Our transition model can capture these patterns. They may even overlap. Like in [13] we track "reads" from other threads (CASE III). In addition we track "writes" from other threads too (CASE IV). Furthermore, we distinguish and recognize "producers" (here: write-instructions in a thread) that invalidate sharers (CASE II and IV).

In the following section we discuss the implementation of our algorithm, the limitations of the implementation and the overall experimental setup.

4.4. Experimental Setup and Limitations

Our algorithm is implemented as a PIN [12] extension. Since we do not require an accurate architectural simulation, we can trade-in accuracy for simulation speed. PIN (version 2.8) is limited to user-space thus we cannot track communication inside or through the operating system[2]. For Parsec this constraint is acceptable due to its computa-

[2]Our concept is independent of the simulator.

tional nature. For server benchmarks, for example, it would not be acceptable and another simulator must be used (e.g. SIMICS). The Parsec authors have already pointed out that PIN induces changes in the thread schedule. Our implementation and experimental setup are susceptible to the same challenge. The conceptual idea, however, is not affected.

We require a machine with sufficient memory (roughly 16GB for an 8-byte chunk size and Parsec) to keep the simulation data in memory. When a program ends then our PIN extension generates and saves all transitions to a text file on disk. For Parsec the biggest text file measured 9GB (152 MB compressed) and the smallest 18kb (740 byte compressed) for an 8 byte chunk size. The two "extremes" were "swaptions" and "blackscholes". A bigger chunk size (> 8 byte) usually reduces the on disk memory requirements drastically. Due to memory constrains we could not simulate "dedup" with 8 byte chunk size (32 byte is feasible however). We used 24 multi-processor XEON-5148 (4way - 2.33GHz) machines with 16GB RAM each. This allowed us to run multiple copies of Parsec simultaneously. We compiled Parsec with "`gcc`" 4.1.2. The simulations were run on a RedHat-5 compatible Linux (kernel 2.6.18). Each workload was executed with the "simlarge" input set for 2, 4, 8, 16, 32, 64, 128 and 256 threads respectively. Our current PIN extension can scale up to 4 cores depending on the workload and program phase. Currently, we use a global lock to access our data structures. It is likely that an improved implementation may be able to scale better.

We recompiled Parsec with the default settings except for one additional compiler switch ("`-g`") in order to attach symbolic information to every executable. This allows us to retrieve function names, source file names and line numbers for every instruction address ("`iaddr`"). For x264 a small additional change was necessary to preserve symbolic information in the executable. In the following section we provide quantitative and qualitative data for Parsec.

5. Experimental Results

In this section we discuss our findings regarding communication in Parsec. First we lay out the limitations of our measurements. Afterwards we analyze how communication scales in Parsec as we increase the number of threads. Then we show how our methodology and tools can be useful to track communication to the source and discover unexpected communication behavior.

5.1. Limitations

As mentioned in Section 4.4 we have used PIN [12] to gather our data. We have noticed that the thread schedule of the underlying operating system can cause variations in the results. This is especially true if (system) processes interfere during simulation. Some applications are more susceptible to distortions than others ("x264" vs. "blackscholes" - for example). Nonetheless, we are confident that our data is sufficiently accurate to draw solid conclusions about communication in Parsec. Especially, since our characterization study of Parsec is the first which does not require an architectural simulation.

The pros of using PIN are that the "simlarge" input set can be simulated to completion for all benchmarks while sweeping the number of threads from 2–256. The "simlarge" input set is already many times smaller than the "native" input set. For a large number of threads (>16) it may be questionable to use smaller input sets than "simlarge".

We have not included "freqmine" in our results due to technical reasons which we are looking into[3]. In "raytrace" we have observed erratic behavior. This may be due to the small input set. For "simlarge" "raytrace" computes 3 frames and for the "native" input set 200[19] in comparison. Due to time- and space-constrains we are missing results for "streamcluster" and "swaptions" above 64- and 128 threads respectively. For 256 threads we noticed an error in "facesim". Thus this value is missing too. For "dedup" we had to increase the granularity (chunk size) from 8 to 32 bytes in order to complete the simulation within our memory resources. All other benchmarks use the system word width granularity (64 bit = 8 byte).

5.2. Communication

Theoretically, the number of connections increases with the number of parallel threads N as $\frac{N^2-N}{2}$ (Eq.1) - if all threads communicate with each other. For N = 2..256 threads we can model (Eq.1) as a power function $f(x) = a \cdot x^b$

[3] "freqmine" is the only application based on OpenMP in Parsec.

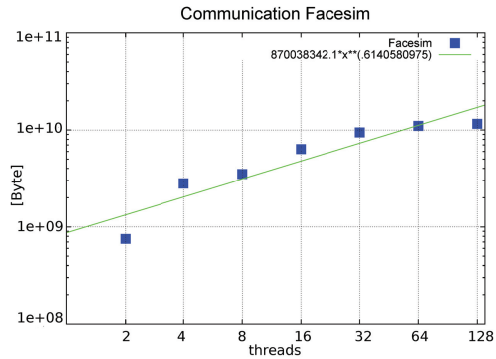

Figure 3: The plot illustrates how the growth of communication (2–256 threads) can be modeled as a power function (green line). The data points represent the amount of data communicated in "facesim". Both axes have a logarithmic scale. Thus the power function can be seen as a straight line.

	black-scholes	facesim	fluid-animate	vips	body-track	ferret	swaptions	canneal	stream-cluster	dedup	x264
abs. amount/ communication	0.0106	0.6141	0.5646	0.2717	0.198	0.1036	1.6395	0.175	0.8353	0.05	-0.145
transitions	0.9534	1.7336	1.2889	1.0244	1.1202	0.3414	2.1183	1.6	1.4563	1.1818	-0.126

Table 1: The table shows the exponent b for communication- and transitions. A value $b = 2$ e.g. expresses a four times increase if the thread count is doubled.

using least square fitting - which yields $b = 2.1135$. Thus our approximation yields a factor of 4.33 (=2^b) for doubling the number of threads ($(a \cdot (2x)^b = 2^b \left[a \cdot x^b\right]$). Figure 3 illustrates how well the growth can be modeled as a power function. Table 1 shows the exponent b for the absolute amount of communication (Figure 5) and the number of transitions between threads (see Figures 1 and 2). The exponent b expresses the relative increase (2^b times) as the number of threads are increased.

Only "swaptions" comes close to the theoretical value ($b = 2.1135$) for the number of transitions. Generally the number of transitions increases more than for communication. "blackscholes", "ferret", "dedup" and "x264" communicate a near constant amount of data from 2–256 threads. "x264" always issues 256 threads - thus no changes were expected. "swaptions" has by far the highest increase in communication: 3.1 times when doubling threads. Communication in "fluidanimate" grows only by the square root in comparison. "blackscholes" scales linearly in regard to the number of transitions. "facesim" and "swaptions" see the biggest increases (3.3x and 4.3x) here.

We also measured the thread connectivity and the number of threads created. A percentage close to 100% means that a thread communicates with (nearly) every other thread. However nothing is said about the amount of communication. For 256 threads we have the following results: "blackscholes" (4.86%), "bodytrack" (99.6%), "canneal" (99.6%), "dedup" (58.2%), "facesim - 128 threads" (99.2%), "ferret" (10.2%), "fluidanimate" (90%), "raytrace" (4.36%), "streamcluster" (53.1%) - 64 threads, "swaptions" (99.2%) - 128 threads, "vips" (14.1%) and "x264" (6.06%). We noticed that "dedup", "ferret" and "vips" have a connectivity close to 100% up to 8, 32 and 64 threads. In "streamcluster" one part of the threads is connected through a star topology, the other part is fully connected (close to 100%). The latter communicates the majority of data. Both "blackscholes" and "x264" contain threads which are connected in a star-topology. In "x264" we additionally see a succession of threads which are connected to a series of related threads. This is probably due to the frame to thread mapping in "x264". Only "dedup" (769) and "ferret" (1027) spawned more than 259 threads.

In Figure 4 we show the growth in communication from 2–256 threads. We took the amount of data communicated between two threads and divided all subsequent values (for 4–256 threads) through it. Thus we can easily see by which factor communication has grown as we increase the number of threads. We further distinguish between "reads", "writes" and "invalidates".

In "blackscholes", "canneal", "dedup" and "vips" read-transitions dominate clearly (CASE III). "bodytrack", "facesim", "ferret" and "x264" experience many invalidations. In "canneal", "ferret", "swaptions" and "vips" we

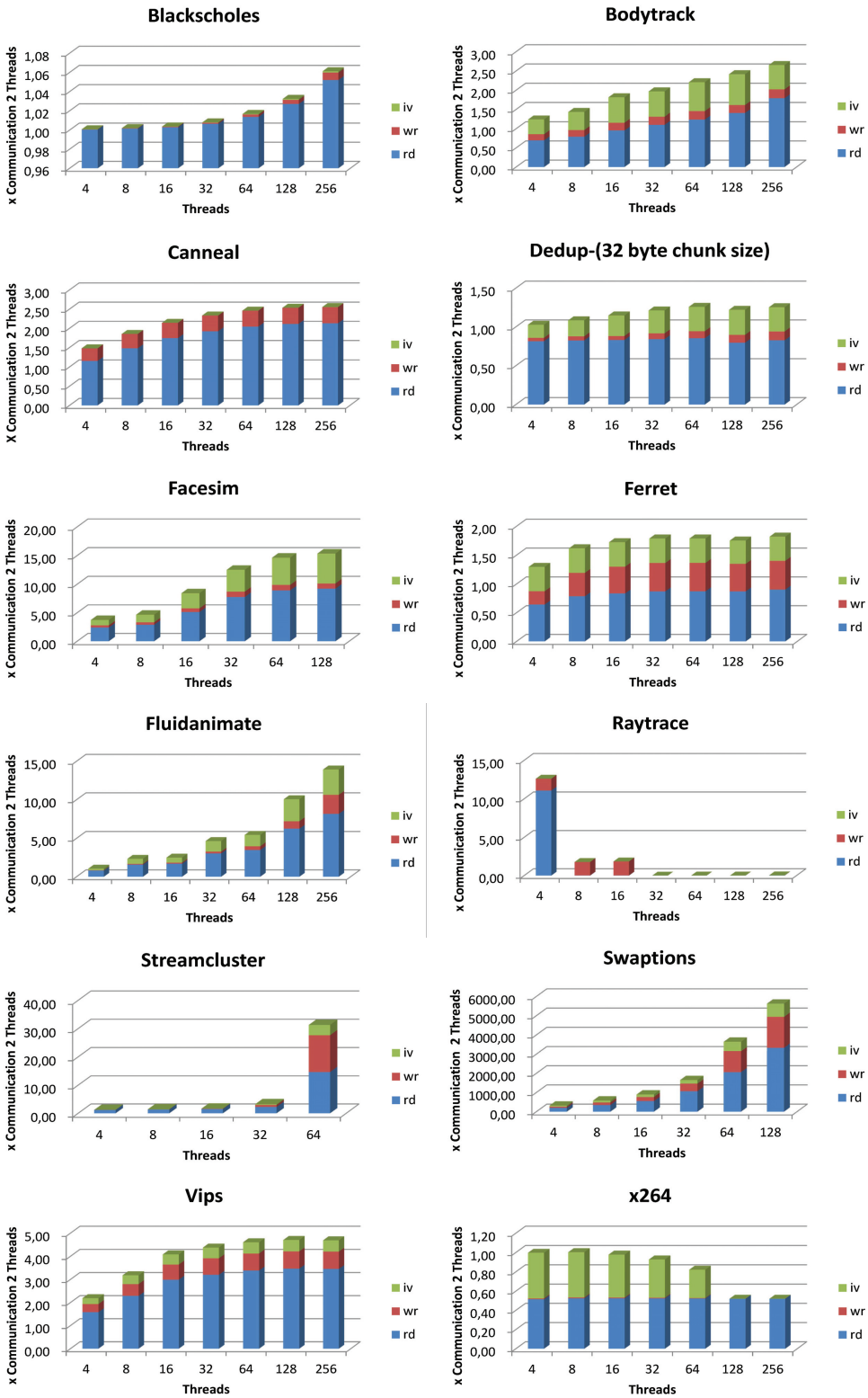

Figure 4: The figures show how much data has been communicated. We distinguish between “reads”, “writes” and “invalidates”. The values are normalized to the 2-thread communication value.

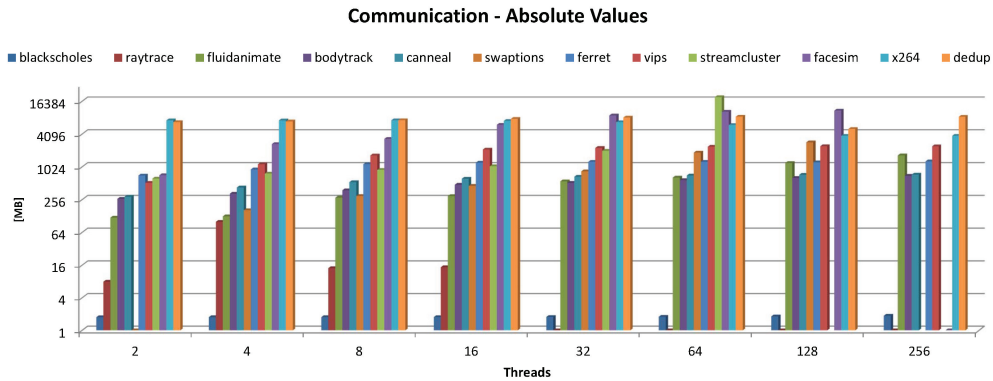

Figure 5: The figure shows the absolute amount of communication in Parsec. The values represent the summed up edge weights of all recorded transitions (read, write and invalidate). Transitions (CASE I-IV) are tracked at machine word granularity (64bit = 8 bytes) - except for "dedup" (32 bytes) since 16 GB RAM and 16GB swap did not suffice to record all transitions at machine word granularity. A few data points are missing due to time- and space-constrains: for "streamcluster" and "swaptions" above 64- and 128 threads respectively. When set to 256 threads, we experienced an error in "facesim". The results for "raytrace" are erratic. We assume that 3 frames (input set "simlarge") are not sufficient to expose sufficient work for more than 4 threads. The "native"-input set in comparison requires 200 frames [19]. Communication in "x264" amounts to ≈3.7 GB from 2–256 threads - if invalidations are factored out (compare with Figure 4). From 64 to 128 threads invalidations drop from 36% to less than 1% in "x264" - since each frame (128 in total) is scheduled to one thread. For "simmedium" (32 frames) invalidations drop at 32 threads. "blackscholes" communicates the least (<2 MB) and "streamcluster" most (19 GB). Surprisingly "streamcluster" jumps from 2 to almost 20 GB for 32- and 64-cores. "swaptions" exerts the biggest increase - from 0.5 MB to 2.8 GB (see Figure 4 as well). Some benchmarks exhibit a steady growth in communication as the number of threads grows ("bodytrack", "facesim", "fluidanimate" and "vips") - others remain relatively constant ("blackscholes", "dedup", "ferret", "canneal", and "x264" without invalidations). The y-axis has a logarithmic scale.

see many "write"-transitions. Here, threads write to memory which was last written to by other threads (CASE IV). Extreme variations can be seen in "raytrace" and "streamcluster". "raytrace" behaves erratically. In "streamcluster" a sudden spike occurs at 64 threads. This requires further investigation. Last but not least, we can see that "x264" gradually experiences fewer invalidations as we reach 128 threads. "x264" dispatches a thread for each frame. "simlarge" has 128 frames - thus invalidations are reduced to a minimum. For "simmedium" (32 frames) we have measured a drop in invalidations at 32 threads.

Figure 5 shows the absolute amount of communication in Parsec. The values represent the summed up edge weights of all recorded transitions (read, write and invalidate). A summary can be found below the figure.

5.3. Tracking Communication to the Source

We have only just started to explore the tracking capabilities of our tools. One of our tools prints out which functions communicate which each and the threads involved. Here an example from "bodytrack" (8-threads):

```
---------------------------------------
RandomGenerator::Rand -> RandomGenerator::Rand 13421744
[2,3,4,5,6,7,8,9->2,3,4,5,6,7,8,9]
---------------------------------------
ParticleFilter::InitializeParticles -> ParticleFilterPthread::Exec 5746712
[1->2,3,4,5,6,7,8,9]
```

We can see that the function "Rand" communicates with itself through different threads ("read"-transition). Roughly 12 MB are communicated. The "InitializeParticles" function communicates with all other threads - a typical initialization pattern. Using our tools we have noticed that "blackscholes" communicates all data only through two lines of code. Other applications like "facesim", for example, communicate at many different places.

6. Conclusion

In Section 4.2 we have explained that our methodology records transitions (CASES I-IV). These transitions are a rich source of information. In this paper we can only analyze and present a tiny fraction. Our initial results have

shown that communication does not increase as one may have expected - even for benchmarks that exhibit all-to-all communication like: “bodytrack”, “canneal”, “facesim”, “swaptions”, “fluidanimate”, “dedup”, “ferret” and “vips” (the latter only up to 8-, 32-, 64-threads). These applications would benefit from an aggressive communication design. “blackschole” has the least communication requirements - a star-, ring- or bus-topology may be sufficient. “x264” has an irregular communication structure. Some of the threads fit onto a high fan-out star- and others onto a mesh-topology.

We have seen that communication scaling in some benchmarks can be modeled as a power-, linear- or constant-function. Only one benchmark (“swaptions”) has shown near theoretical behavior in the case of all-to-all communication. Last but not least our methodology and tools reveal communication on source-level which may be interesting for compiler architects.

Acknowledgments

We would like to thank Fazal Hameed and Avinash Pathak for their comments and proofreading.

References

[1] R. Balasubramonian, N. Muralimanohar, K. Ramani, L. Cheng, J. Carter, Leveraging wire properties at the microarchitecture level, Micro, IEEE 26 (6) (2006) 40 –52. doi:10.1109/MM.2006.123.

[2] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown, Mibench: A free, commercially representative embedded benchmark suite, Workload Characterization, Annual IEEE International Workshop 0 (2001) 3–14. doi:http://doi.ieeecomputersociety.org/10.1109/WWC.2001.15.

[3] C. Lee, M. Potkonjak, W. H. Mangione-Smith, Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems, in: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture, MICRO 30, IEEE Computer Society, Washington, DC, USA, 1997, pp. 330–335.
URL http://portal.acm.org/citation.cfm?id=266800.266832

[4] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, The splash-2 programs: characterization and methodological considerations, SIGARCH Comput. Archit. News 23 (1995) 24–36. doi:http://doi.acm.org/10.1145/225830.223990.
URL http://doi.acm.org/10.1145/225830.223990

[5] C. Bienia, S. Kumar, K. Li, Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors, in: Proceedings of the 2008 International Symposium on Workload Characterization, 2008.

[6] C. Bienia, S. Kumar, J. P. Singh, K. Li, The parsec benchmark suite: Characterization and architectural implications, in: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, 2008.

[7] C. Bienia, Benchmarking modern multiprocessors, Ph.D. thesis, Princeton University (January 2011).

[8] C. Bienia, K. Li, Fidelity and scaling of the parsec benchmark inputs, in: Proceedings of the 2010 International Symposium on Workload Characterization, 2010.

[9] A. Jaleel, M. Mattina, B. Jacob, Last level cache (llc) performance of data mining workloads on a cmp - a case study of parallel bioinformatics workloads, High-Performance Computer Architecture, International Symposium on 0 (2006) 88–98. doi:http://doi.ieeecomputersociety.org/10.1109/HPCA.2006.1598115.

[10] M.-L. Li, R. Sasanka, S. Adve, Y.-K. Chen, E. Debes, The alpbench benchmark suite for complex multimedia applications, IEEE Workload Characterization Symposium 0 (2005) 34–45. doi:http://doi.ieeecomputersociety.org/10.1109/IISWC.2005.1525999.

[11] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, M. Yarrow, The nas parallel benchmarks 2.0, Report NAS-95-020 0.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, SIGPLAN Not. 40 (2005) 190–200. doi:http://doi.acm.org/10.1145/1064978.1065034.
URL http://doi.acm.org/10.1145/1064978.1065034

[13] N. Barrow-Williams, C. Fensch, S. Moore, A communication characterization of splash-2 and parsec, in: Proceedings of the 2009 International Symposium on Workload Characterization, 2009.

[14] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, S. K. Reinhardt, The m5 simulator: Modeling networked systems, IEEE Micro 26 (2006) 52–60. doi:http://doi.ieeecomputersociety.org/10.1109/MM.2006.82.

[15] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, H. Angepat, Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators, in: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40, IEEE Computer Society, Washington, DC, USA, 2007, pp. 249–261. doi:http://dx.doi.org/10.1109/MICRO.2007.16.
URL http://dx.doi.org/10.1109/MICRO.2007.16

[16] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, P. yves Droz, Ramp blue: a message-passing manycore system in fpgas, in: In 2007 International Conference on Field Programmable Logic and Applications, FPL 2007, 2007, pp. 27–29.

[17] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, K. Asanović, Ramp: Research accelerator for multiple processors, IEEE Micro 27 (2007) 46–57. doi:http://doi.ieeecomputersociety.org/10.1109/MM.2007.39.

[18] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, A. lsK., Ramp gold: An fpga-based architecture simulator for multiprocessors, in: Design Automation Conference (DAC), 2010 47th ACM/IEEE, 2010, pp. 463 –468.

[19] C. Bienia, K. Li, Parsec 2.0: A new benchmark suite for chip-multiprocessors, in: Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation, 2009.